Apuntes Java - Stream

Profesor: Luis Javier López López

Índice

Stream

Introducción

En Java, un Stream (flujo) es una secuencia de elementos que puede ser procesada de manera declarativa. Los Stream fueron introducidos en Java 8 como parte del paquete java.util.stream y proporcionan una forma más eficiente y expresiva de realizar operaciones en colecciones de datos, como listas, conjuntos y mapas.

Aquí hay algunas características clave de los Stream:

- **Secuencia de elementos**: Un Stream representa una secuencia de elementos. Puede ser una secuencia de números, cadenas, objetos o cualquier tipo de dato.
- **Operaciones de alto nivel**: Los Stream permiten realizar operaciones de alto nivel en los datos de manera declarativa. Esto significa que puedes expresar lo que deseas hacer con los datos, en lugar de especificar cómo hacerlo mediante bucles explícitos.
- **Operaciones intermedias y terminales**: Las operaciones en un Stream se pueden dividir en dos tipos: operaciones intermedias y operaciones terminales. Las operaciones intermedias, como ***filter***, ***map*** y ***sorted***, son transformaciones que crean un nuevo Stream. Las operaciones terminales, como ***forEach, reduce y collect***, producen un resultado final o ejecutan una acción en los elementos del Stream.
- **Lazy evaluation:** Una característica importante de los Stream es que utilizan la evaluación perezosa (lazy evaluation). Esto significa que las operaciones no se ejecutan hasta que sea necesario obtener un resultado final. Esto puede conducir a una mayor eficiencia, ya que solo se procesan los elementos que son necesarios.
- **Pipelines de operaciones:** Puedes encadenar varias operaciones en un Stream para formar un "pipeline" de operaciones. Esto te permite realizar una secuencia de transformaciones y filtrados en los datos de manera concisa.

Aquí hay un ejemplo simple para ilustrar el uso de Stream:

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int suma = numeros.stream()
                  .filter(n -> n % 2 == 0)
                  .mapToInt(Integer::intValue)
                  .sum();

System.out.println("Suma de números pares: " + suma);
```

Construcción de expresiones

Los pasos clásicos para construir una expresión con Stream a partir de una colección en Java son los siguientes:

Paso 1: Obtener un Stream desde la colección

Empiezas con una colección existente, como una lista, y obtienes un Stream a partir de ella utilizando el método stream():

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Stream<Integer> stream = numeros.stream();
```

Paso 2: Aplicar operaciones intermedias

En este paso, aplicamos operaciones intermedias al Stream para transformar, filtrar o manipular los datos según tus necesidades. Por ejemplo, puedes filtrar los números pares y elevar al cuadrado cada número:

```
Stream<Integer> resultadoStream = stream
    .filter(n -> n % 2 == 0)  // Filtrar números pares
    .map(n -> n * n);         // Elevar al cuadrado
```

Paso 3: Aplicar una operación terminal

Una operación terminal es necesaria para obtener un resultado final. Puedes elegir entre varias operaciones terminales, dependiendo de lo que necesites. En este ejemplo, vamos a recoger los resultados en una lista utilizando collect:

```
List<Integer> resultadoFinal = resultadoStream.collect(Collectors.toList());
```

Expresión final completa

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
List<Integer> resultadoFinal = numeros.stream()
    .filter(n -> n % 2 == 0)
    .map(n -> n * n)
    .collect(Collectors.toList());
```

Funciones Lambda

Las expresiones lambda son una característica clave de Java que te permite crear instancias concisas de interfaces funcionales, como Comparator, Predicate, Consumer, Function , etc. Estas interfaces funcionales son interfaces que contienen un único método abstracto, y son usadas extensivamente en el contexto de las expresiones lambda.

Comparator (Interfaz Funcional para ordenar)

```
// Sintaxis: (parámetro1, parámetro2) -> expresión o bloque de código

// Ejemplo para comparar dos números de manera ascendente
Comparator<Integer> ascendente = (num1, num2) -> num1.compareTo(num2);

// Ejemplo para comparar dos números de manera descendente
Comparator<Integer> descendente = (num1, num2) -> num2.compareTo(num1);
```

Predicate(Interfaz Funcional para condiciones)

```
// Sintaxis: (parámetro) -> expresión booleana o bloque de código que devuelve
un booleano

// Ejemplo para verificar si un número es par
Predicate<Integer> esPar = num -> num % 2 == 0;

// Ejemplo para verificar si un String tiene más de 5 caracteres
Predicate<String> longitudMayor5 = str -> str.length() > 5;
```

Consumer (Interfaz Funcional para consumir elementos):

```
// Sintaxis: (parámetro) -> bloque de código que realiza alguna acción

// Ejemplo para imprimir un número
Consumer<Integer> imprimirNumero = num -> System.out.println(num);

// Ejemplo para imprimir un String en mayúsculas
Consumer<String> imprimirMayusculas = str -> System.out.println(str.toUpperCase());
```

Function (Interfaz Funcional para transformar elementos):

```
// Sintaxis: (parámetro) -> expresión o bloque de código que realiza una transformación
y devuelve un resultado

// Ejemplo para elevar al cuadrado un número
Function<Integer, Integer> cuadrado = num -> num * num;

// Ejemplo para concatenar "Hola, " a un String
Function<String, String> agregarSaludo = str -> "Hola, " + str;
```

Operaciones Intermedias

Estas son algunas de las operaciones intermedias más comunes proporcionadas por la api Stream:

- **filter(Predicate<T> predicate):** Filtra los elementos del Stream según la condición especificada en el Predicate.
- **map(Function<T, R> mapper):** Transforma cada elemento del Stream utilizando la función proporcionada por el mapper.
- **flatMap(Function<T, Stream<R>> mapper):** Similar a map, pero puede generar cero, uno o varios elementos para cada elemento de entrada**.**
- **distinct():** Elimina elementos duplicados del Stream basándose en su implementación de equals.
- **sorted():** Ordena los elementos del Stream en orden natural**.**
- **peek(Consumer<T> action):** Proporciona una manera de realizar operaciones adicionales en cada elemento del Stream sin modificarlos.
- **limit(long maxSize):** Limita el tamaño del Stream a un número máximo de elementos.
- **skip(long n):** Omite los primeros n elementos del Stream.
- **takeWhile(Predicate<T> predicate):** Devuelve elementos del Stream mientras la condición especificada sea verdadera.
- **dropWhile(Predicate<T> predicate):** Elimina elementos del Stream mientras la condición especificada sea verdadera.
- **filter(Predicate<T> predicate):** Filtra los elementos del Stream según la condición especificada en el Predicate.
- **mapToDouble(ToDoubleFunction<T> mapper), mapToInt(ToIntFunction<T> mapper), mapToLong(ToLongFunction<T> mapper):** Variantes especializadas de map para tipos primitivos que devuelven flotantes, enteros o longs.
- **boxed():**Convierte un Stream de tipos primitivos en un Stream de objetos envolventes.
- **parallel():** Retorna un Stream paralelo, permitiendo operaciones en paralelo para un mejor rendimiento en conjuntos de datos grandes.
- **sequential():** Convierte un Stream paralelo en un Stream secuencial.
- **unordered():**Retorna un Stream sin garantía del orden de los elementos.

Operaciones Finales

Estas son algunas de las operaciones finales más comunes proporcionadas por la api Stream:

- **forEach(Consumer<T> action):** Realiza una acción para cada elemento del Stream. Utilizado para realizar operaciones en cada elemento, como imprimirlos o aplicar acciones específicas.
- **toArray():** Convierte los elementos del Stream en un array. Útil cuando necesitas trabajar con los datos como un array.
- **reduce(BinaryOperator<T> accumulator):** Combina los elementos del Stream en un solo resultado mediante una operación de reducción. Puede ser utilizado para sumar, multiplicar, encontrar el máximo, entre otros.
- **collect(Collector<T, A, R> collector):**Realiza una operación de acumulación utilizando el Collector proporcionado. Útil para transformar los elementos del Stream en una estructura de datos específica, como una lista, conjunto o mapa.
- **min(Comparator<T> comparator), max(Comparator<T> comparator):**Encuentra el elemento mínimo o máximo según el comparador proporcionado. Utilizado para encontrar el valor mínimo o máximo en un Stream.
- **count():**Cuenta el número de elementos en el Stream. Proporciona la cantidad total de elementos en el Stream.
- **anyMatch(Predicate<T> predicate), allMatch(Predicate<T> predicate), noneMatch(Predicate<T> predicate):**Verifican si algún, todos o ninguno de los elementos cumplen con la condición especificada. Son operaciones de evaluación de predicados.
- **findFirst(), findAny():** Encuentran el primer elemento del Stream. findAny es más eficiente en operaciones paralelas. Utilizados para obtener el primer elemento que cumple con ciertas condiciones.
- **toList(), toSet(), toMap():** Convierten los elementos del Stream en una lista, un conjunto o un mapa. Útiles para transformar los resultados del Stream en estructuras de datos específicas.

Ejemplos

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class EjemplosStreamAdicionales {

    public static void main(String[] args) {
        // Crear una lista de palabras
        List<String> palabras = Arrays.asList("java", "stream", "api", "lambda", "ejemplo","programacion");

        // Ejemplo 1: Filtrar palabras que contienen la letra 'a' y convertirlas a mayúsculas
        List<String> palabrasConAEnMayusculas = palabras.stream()
                .filter(palabra -> palabra.contains("a"))
                .map(String::toUpperCase)
                .collect(Collectors.toList());

        System.out.println("Palabras con 'a' en mayúsculas: " + palabrasConAEnMayusculas);

        // Ejemplo 2: Encontrar la longitud promedio de las palabras
        double longitudPromedio = palabras.stream()
                .mapToInt(String::length)
                .average()
                .orElse(0.0);

        System.out.println("Longitud promedio de las palabras: " + longitudPromedio);

        // Ejemplo 3: Verificar si alguna palabra tiene más de 7 caracteres
        boolean algunaConMasDe7Caracteres = palabras.stream()
                .anyMatch(palabra -> palabra.length() > 7);

        System.out.println("¿Alguna palabra tiene más de 7 caracteres? " + algunaConMasDe7Caracteres);

        // Ejemplo 4: Concatenar todas las palabras con un guión entre ellas
        String concatenacionConGuion = palabras.stream()
                .reduce((palabra1, palabra2) -> palabra1 + "-" + palabra2)
                .orElse("");

        System.out.println("Concatenación con guión: " + concatenacionConGuion);

        // Ejemplo 5: Encontrar la palabra más larga
        String palabraMasLarga = palabras.stream()
                .max((palabra1, palabra2) -> Integer.compare(palabra1.length(), palabra2.length()))
                .orElse("");

        System.out.println("Palabra más larga: " + palabraMasLarga);
    }
}
```

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class EjemplosStream {

    public static void main(String[] args) {
        // Crear una lista de números
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        // Ejemplo 1: Filtrar y transformar elementos
        List<Integer> paresAlCuadrado = numeros.stream()
                .filter(n -> n % 2 == 0)           // Filtrar números pares
                .map(n -> n * n)                   // Elevar al cuadrado
                .collect(Collectors.toList());     // Recoger en una lista

        System.out.println("Pares al cuadrado: " + paresAlCuadrado);

        // Ejemplo 2: Encontrar el doble del primer número par
        int doblePrimerPar = numeros.stream()
                .filter(n -> n % 2 == 0)
                .mapToInt(n -> n * 2)              // Doble del número par
                .findFirst()                       // Obtener el primer resultado
                .orElse(0);                        // Valor por defecto si no se encuentra

        System.out.println("Doble del primer número par: " + doblePrimerPar);

        // Ejemplo 3: Verificar si todos los números son mayores que 5
        boolean todosMayoresACinco = numeros.stream()
                .allMatch(n -> n > 5);

        System.out.println("¿Todos los números son mayores que 5? " + todosMayoresACinco);

        // Ejemplo 4: Imprimir cada número
        numeros.stream()
                .forEach(System.out::println);

        // Ejemplo 5: Obtener la suma de los cuadrados de los números impares
        int sumaCuadradosImpares = numeros.stream()
                .filter(n -> n % 2 != 0)           // Filtrar números impares
                .mapToInt(n -> n * n)              // Elevar al cuadrado
                .sum();                            // Sumar

        System.out.println("Suma de los cuadrados de los impares: " + sumaCuadradosImpares);
    }
}
```